DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

20020411 086

A Decision Procedure for Bit-Vector Arithmetic

Clark W. Barrett, David L. Dill, and Jeremy R. Levitt*
Computer Systems Laboratory, Stanford University
Stanford, CA 94305, USA

Abstract

Bit-vector theories with concatenation and extraction have been shown to be useful and important for hardware verification. We have implemented an extended theory which includes arithmetic. Although deciding equality in such a theory is NP-hard, our implementation is efficient for many practical examples. We believe this to be the first such implementation which is efficient, automatic, and complete.

1 Introduction

As designs grow in complexity, design verification becomes increasingly important and challenging. New and better verification techniques are critical to ensure correctness, maintain design cycle times, and protect designers from economic losses due to undiscovered bugs. Formal methods for verification are especially attractive because they have the potential to cover most or all of the behaviors in a design without having to exhaustively simulate it.

The Stanford Validity Checker (SVC) [2, 9] is an automatic verification tool which has been in development for several years at Stanford University. The input to SVC is a Boolean formula in a quantifier-free subset of first-order logic. It may also contain Boolean operators, uninterpreted functions, and various interpreted functions such as operations on infinite arrays and arithmetic. We have found these constructs to be useful for modeling hardware designs. Using a combination of case-splitting and cooperating decision procedures, SVC determines whether a formula is valid (i.e. equivalent to true in every possible interpretation). If the formula is not valid, SVC returns a counterexample. SVC is used as the final step in the automatic hardware verification paradigm of Burch and Dill [4]. In their approach a specification and an implementation are each symbolically simulated and the resulting states are then compared to see if they are equivalent. This method has been shown to be successful for verification of actual designs and is currently being applied to the TORCH microprocessor, an aggressive superscalar microprocessor developed for educational and research purposes at Stanford University [11]. The powerful and efficient decision procedures in SVC are critical for the success of this effort.

Other formal methods such as theorem proving and model checking have been used extensively, but theorem provers suffer from a lack of automation and model checking from the inability to handle large designs due to state explosion. SVC attempts to take the best (and avoid the worst) of both worlds. First of all, SVC is able to reason at an abstract level and has built-in decision procedures much like a theorem prover. However, the logic of SVC is restricted to be decidable, which enables it to prove or disprove all formulas automatically. Secondly, SVC uses a directed acyclic graph (DAG) structure much like that used for binary decision diagrams (BDDs) in model checkers. However, SVC does not require the DAG to be canonical. As a result, SVC is more robust and can handle formulas that would blow up if represented with BDDs. Such formulas may still take a very long time to verify, but various heuristics can be applied to speed up the verification without fear of a sudden failure due to BDD explosion.

As mentioned, a nice feature of theorem provers is their support for abstraction. A primary goal of SVC is to make abstraction easier by providing uninterpreted functions and various interpreted theories. Most recently, we have completed a decision procedure for a theory of bit-vectors in SVC. Bit-vectors (also called "words") are a critical abstraction for reasoning about hardware structures. Intuitively, a bit-vector is a fixed-length string of individual bits, and operations on bit-vectors can be described in terms of their effect on each bit. Alternatively, these operations can be viewed as transformations on bit-vectors as a whole. The advantage of the latter approach is that a property which may be complex and difficult at the bit-level (such as addition) can be expressed easily as an operation on bit-vectors. It is especially desirable to be able to reason about concatenation, extraction, bit-wise Boolean operations, and arithmetic, since these correspond to hardware structures.

As we describe in the next section, deciding equality of arbitrary combinations of these operations is NP-hard. In spite of this fact, we have developed an automatic algorithm for reasoning about fixed-size bit-vector addition, negation, concatenation, and extraction which avoids exponential blow-up on many practical examples.[1]

As mentioned above, the approach taken in SVC differs from that of both theorem provers and model checkers. However, there is closely related work in both areas. Recently, successful methods for reasoning about bit-vector operations in a model-checking paradigm have used Binary Moment Diagrams (*BMDs) [1, 3]. These have been able to automatically verify large arithmetic circuits. The set of problems solvable using *BMDs is comparable to those solvable by SVC's bit-vector canonizer and a comparison of the two is presented in Section 5 below.

Bit-vector libraries have also been developed for many theorem provers including Boyer-Moore [8], SDVS [10], HOL [13] and PVS [6]. All of these libraries implement the basic operations of concatenation and extraction, but none of

*Now at 0-in Design Automation.

Draft: to appear in the 35th Design Automation Conference (DAC), June 1998, San Francisco, California.

[1] Although SVC is capable of representing other Boolean bit-vector operations, we will not discuss them in this paper.

$x_{[n]}$	A bit-vector of size n. We will sometimes omit the subscript if it is obvious from the context.
$\mathtt{val}_{[n]}$	A constant bit-vector which is the binary representation of the decimal value `val`. If n is larger than is required to represent `val`, then the upper bits are assumed to be 0. If n is omitted, it is assumed to be the smallest value required to represent `val`. We only use values which are positive, except in the case of $(-1)_{[n]}$ which we use to represent the vector of size n containing all 1's (i.e. having value $2^n - 1$).
$x_{[n]}[i:j]$	The extraction of bits i through j of x. We require $0 \le j \le i < n$. We write $x[i]$ as an abbreviation when $i = j$.
$x_{[m]} \circ y_{[n]}$	The concatenation of x and y to yield a new bit-vector of size $m + n$.
NOT $x_{[n]}$	The bit-vector whose bits are the negation of the bits of x.
$x_{[m]} +_{[k]} y_{[n]}$	Addition of $x_{[m]}$ and $y_{[n]}$ modulo 2^k. If k is omitted, we assume it is equal to the larger of m and n. Also, if m is less than k, then x is implicitly zero-extended to size k, whereas if $k < m$, the intended meaning is that only the lowest k bits of x are to be used (and similarly for n and y). Because modular addition is associative, it is unnecessary to use parentheses when referring to more than two operands.
$x_{[m]} = y_{[n]}$	True if and only if $m = n$ and corresponding bits of x and y are equal.

Table 1: Bit-vector Theory and Definitions

them provide a complete and automatic implementation of bit-vector arithmetic. Probably the most closely related work is that of Cyrluk et al. in PVS. A comparison of of their work with SVC is included in Section 4 below.

The rest of the paper is organized as follows. Section 2 describes some notation and complexity results. Section 3 explains in some detail the theory behind the SVC implementation and contains the main contributions of the paper. Section 4, as mentioned, contains a comparison of our method with that presented in [6], and Section 5 gives experimental results obtained using SVC on microprocessor verification examples. Finally, Section 6 gives some conclusions and directions for future work.

2 Complexity of Bit-Vector Logics

In contrast to those approaches which convert bit-vectors into natural numbers, our approach is to remain in the bit-vector domain; all operations, therefore, take bit-vectors as arguments and return bit-vectors as results. Table 1 lists the elements that make up our theory of bit-vectors. Note that we refer to the bits in a bit-vector of size n by index, with the least significant (right-most) bit being indexed with 0 and the most significant bit being indexed with $n - 1$.

In order to evaluate our decision procedure it is necessary to answer the question of whether there exists a simple algorithm for deciding expressions in this bit-vector theory. Some complexity results are given in [6]. It is shown that equality of terms under the core theory of fixed-size bit-vectors with concatenation and extraction is decidable in polynomial time. A subsequent extension to include bit-vector Boolean operations such as **AND**, **OR** and **NOT**, however, can easily be shown to produce a theory in which deciding equality is NP-hard as follows. Consider an arbitrary instance of the Boolean satisfiability problem which is a well-known NP-complete problem. A general Boolean proposition can be encoded using 1-bit bit-vectors and the Boolean operators. Call this encoding P. Then the satisfiability problem can be solved by checking the validity of $P = FALSE$. If valid, the formula is unsatisfiable, otherwise it is satisfiable.

Alternatively, consider extending the core theory by including arithmetic operations. Unfortunately, even the most trivial extension can quickly be seen to be NP-hard. Allowing only the additional operation of adding one to any bit-vector immediately gives us the ability to express arbitrary propositional logic statements since

$$\begin{array}{lcll} x_{[1]} + 1 & = & \mathbf{NOT}\ x_{[1]} & \text{and} \\ (x_{[1]} \circ y_{[1]} +_{[3]} 1)[2] & = & x_{[1]}\ \mathbf{AND}\ y_{[1]}. & \end{array}$$

The same reduction as above shows that deciding equality in this simple extended theory is NP-hard. This gives some insight into the difficulty of handling bit-vector arithmetic automatically.

Finally, one additional complication when dealing with bit-vectors is that there are some bit-vector formulas which are valid only because each bit-vector variable has a finite number of possible values. For example,

$$a_{[1]} = b_{[1]} \vee b_{[1]} = c_{[1]} \vee a_{[1]} = c_{[1]}.$$

In order to correctly handle these formulas, we must manually force SVC to consider all possible values for each variable. Fortunately, the examples we have encountered in practice do not have this property.

3 Our Approach

As described in [2], SVC uses a framework for cooperating decision procedures very much like that developed by Shostak [5] and used in PVS. One of the requirements of this framework is that semantically equivalent terms should have a unique representation, which we refer to as a *canonical* form. We call the process of transforming terms into their canonical form *canonizing* and we call the algorithm which does it a *canonizer*. In SVC, not all terms need to be canonized. Only terms which do not contain Boolean subexpressions, which we call *atomic*, must be maintained in canonical form. This policy is acceptable because non-atomic expressions contain at least one term which SVC can use to perform a case-split. Since SVC exhausts all possible case-splits before reporting a counterexample, it is impossible for a false negative to result. As we will see below, the flexibility of not having to canonize non-atomic expressions can be exploited to delay canonization of complicated expressions until absolutely necessary. Our framework further requires that atomic equations be written in a specific form in which the left-hand side contains a single variable and the right-hand side containes the rest of the terms. We call the algorithm to do this a *solver*. Every time a new theory is added to SVC, a canonizer and solver for that theory must be provided. The canonizer and solver for bit-vectors in

(1)	$\alpha_{[m]} \circ \beta_{[n]}$	$\rightarrow$	$2^n \cdot \alpha_{[m]} +_{[m+n]} \beta_{[n]}$	
(2)	**NOT** $\alpha_{[m]}$	$\rightarrow$	$(-1)_{[m]} \cdot \alpha_{[m]} +_{[m]} (-1)_{[m]}$	
(3)	$w_{[m]}[i:j]$	$\rightarrow$	$y_{[i-j+1]}$ Additionally, the following equation is added to the current knowledge database : $w_{[m]} = 2^{(i+1)} \cdot x_{[m-i-1]} +_{[m]} 2^j \cdot y_{[i-j+1]} +_{[m]} z_{[j]}$ where x, y, and z are new variables. If $i = m-1$ or $j = 0$ then the appropriate terms are omitted.	
(4)	$(x_0 +_{[m]} \cdots x_s)[i:0]$	$\rightarrow$	$x_0 +_{[i+1]} \cdots x_s$	
(5)	$(x_0 +_{[m]} \cdots x_s)[i:j]$	$\rightarrow$	$x_0[i_0:j] +_{[i-j+1]} \cdots x_s[i_s:j] +_{[i-j+1]}$ $\mathbf{OVF}_{[i:j]}(x_0[j-1:0] \ldots x_k[j-1:0])$	if $j > 0$
(6)	$\alpha +_{[i]} (x_0 +_{[j]} \cdots x_s)$	$\rightarrow$	$\alpha +_{[i]} x_0 +_{[i]} \cdots x_s$	if $j \geq i$
(7)	$\alpha +_{[i]} (x_0 +_{[j]} \cdots x_s)$	$\rightarrow$	$\alpha +_{[i]} x_0 +_{[i]} \cdots x_s +_{[i]} 2^j \cdot (-1)_{[i-j]} \cdot \mathbf{OVF}_{[i-1:j]}(x_0 \ldots x_s)$	if $j < i$
(8)	$\mathbf{OVF}_{[i:j]}(\alpha_0 \ldots \alpha_k)$	$\rightarrow$	$\mathbf{OVF}_{[i:j]}(\alpha_0, (\alpha_1 +_{[j]} \cdots \alpha_k)) +_{[i-j+1]} \mathbf{OVF}_{[i:j]}(\alpha_1 \ldots \alpha_k)$	if $k > 1$
(9)	$\mathbf{OVF}_{[i:j]}(\alpha_{[m]}, \beta_{[n]})$	$\rightarrow$	$\alpha[m-1:j] +_{[i-j+1]} \beta[n-1:j] +_{[i-j+1]}$ $\mathbf{OVF}_{[j:j]}(\alpha[j-1:0], \beta[j-1:0])$	if $i > j$
(10)	$\mathbf{OVF}_{[n:n]}(\alpha_{[n]}, \beta_{[n]})$	$\rightarrow$	$\mathbf{OVF}_{[n-1:n-1]}(\mathbf{ite}(\alpha[n-1] = \beta[n-1]\ ,\ \alpha[n-1] \circ 0_{[n-2]}\ ,\ \alpha[n-2:0]),$ $\mathbf{ite}(\alpha[n-1] = \beta[n-1]\ ,\ \beta[n-1] \circ 0_{[n-2]}\ ,\ \beta[n-2:0]))$	if $n > 1$
	$\mathbf{OVF}_{[n:n]}(\alpha_{[n]}, \beta_{[n]})$	$\rightarrow$	$\mathbf{ite}(\alpha = \beta\ ,\ \alpha\ ,\ 0_{[1]})$	if $n = 1$

Table 2: Rules for eliminating concatenation, negation, and extraction, flattening addition, and converting **OVF** terms to non-atomic expressions. Note that α and β are arbitrary expressions and $i_k = min(i, n_k)$ where n_k is the size of x_k.

SVC are based on properties of hardware arithmetic. They constitute the major contribution of this paper and are described in the next two subsections.

3.1 Canonizer

Coming up with a canonical form for bit-vector expressions is complicated by the inclusion of bit-vector arithmetic. This is because the same expression may be represented in non-trivially different ways. For example, $(x_{[n]} +_{[n+1]} x_{[n]})$ is equivalent to $(x_{[n]} \circ 0_{[1]})$. Similarly, $(x_{[1]} +_{[1]} 1_{[1]})$ is equivalent to (**NOT** $x_{[1]}$).

To avoid such redundancy, we translate all bit-vector expressions into a specific kind of arithmetic expression: the addition (modulo 2^n for some fixed bit-width n) of bit-vector variables with constant coefficients. We call these ***bitplus*** expressions. In order to ensure a unique representation, variables are ordered with duplicates eliminated, and each coefficient is reduced modulo 2^n. A set of transformations for converting bit-vector operations into bitplus expressions is shown in Table 2. Some of these rules make use of the **OVF** operator, which we define and explain below. The first two rules are simple transformations for dealing with concatenation and negation. Rule (3) shows how to eliminate extraction at the cost of introducing new variables. We refer to this process as ***slicing***, and it is desirable to avoid it whenever possible. Intuitively, repeated slicing moves from the bit-vector abstraction to the bit-level, and in the worse case, each bit must be considered. Section 4 provides one illustration of how slicing can be avoided. Rules (4) and (5) show how to eliminate extraction when applied to a bitplus expression, and (6) and (7) show how to "flatten" bitplus expressions to ensure that other bitplus expressions do not appear as subexpressions.

However, a canonical form cannot always be obtained by simple transformations (as we would expect given the fact that the general problem is NP-hard). The difficulty comes from the interaction of extraction and addition. Consider the following two expressions.

$$(8 \cdot x_{[n]} +_{[4]} 7)[2]$$

$$(x_{[3]} +_{[4]} y_{[3]})[3]$$

In the first case, it is desirable to push the extraction inside the bitplus expression which will result in $1_{[1]}$. However, in the second case, there is no way to represent the result of pushing the extraction inside the bitplus, because the result depends on whether adding x and y overflows into the fourth (most significant) bit. To deal with such cases, we introduce a new *overflow* operator, $\mathbf{OVF}_{[i:j]}$ which represents bits i through j of the sum of its operands. Using this operator, we can rewrite the second expression above as

$$(x_{[3]} +_{[4]} y_{[3]})[3] = \mathbf{OVF}_{[3:3]}(x, y).$$

We define overflow for the general case as follows:

$$\mathbf{OVF}_{[i:j]}(x_{0[n_0]} \ldots x_{k[n_k]}) = (x_{0[n_0]} +_{[i+1]} \cdots x_{k[n_k]})[i:j].$$

Whenever the overflow operator is applied, the expression is first checked using a simple algorithm to see if it is equivalent to a concatenation of variables or their negations. If it is, then the appropriate bits are extracted and then converted back into a bitplus expression, for example,

$$\mathbf{OVF}_{[3:1]}(2 \cdot y_{[2]}, 9) = y_{[2]} + 4.$$

If the overflow expression cannot be written in a simple form as above, we break it down using the last three rules shown in Table 2.

To understand rule (8), notice that $\mathbf{OVF}_{[i:j]}(x, y, z)$ and $\mathbf{OVF}_{[i:j]}(x, (y +_{[j]} z))$ differ by exactly $\mathbf{OVF}_{[i:j]}(y, z)$. Rule (8) is the generalization of this property which we use to split an overflow expression with k arguments into two overflow expressions that have 2 and $k-1$ arguments respectively. This rule is applied repeatedly until all overflow expressions have only two arguments. Then rule (9) is applied to convert overflow expressions in which $i > j$ to overflow expressions in which $i = j$ (we show the case where $m > j$ and $n > j$, but the other cases are similar). Finally, rule (10) takes overflow expressions resulting from the application of rule (9) and turns them into non-atomic expressions. It does this by making use of the **ite** operator which is used in SVC to represent all Boolean operations. For arbitrary expressions, α, β, and γ,

$$\textbf{ite}(\alpha, \beta, \gamma) = \textbf{if } \alpha \textbf{ then } \beta \textbf{ else } \gamma.$$

The intuition behind rule (10) is that we are simply computing the carry bit of an n-bit adder. If the most significant bits are equal, then they determine and are equal to the carry bit. If they are different, then the carry bit is propagated from other $n-1$ bits. The use of the **ite** operator is of key importance. As we mentioned earlier, SVC does not require a canonical form for non-atomic expressions. A canonical form would require expressing the full logic of a ripple-carry adder and would require looking at all the bits of the arguments. But the decomposition we have given is incremental, suspended until SVC does a case-split on the equality of the most significant bits. If they are unequal, SVC will slice off the next most significant bits. In the worst case, we will have to look at every bit. But the incremental approach will avoid this unless absolutely necessary.

3.2 Solver

The other main contribution of this paper is a solver for equations involving bit-vector operations. The requirements for the solver are very similar to those of the canonizer. In fact, the solver can be viewed as a canonizer for equations. In SVC, canonical equations are required to have the most complex variable or uninterpreted function isolated on the left-hand side, with the rest of the terms on the right-hand side. Complexity is defined by a total ordering on expressions (see [2]). In the case of bit-vectors, we arrange for longer bit-vectors to be more complex than shorter ones, so that we solve for the longest bit-vector in the equation. This avoids slicing bit-vectors unnecessarily. In general, we must be able to solve arbitrary equations of the form

$$a_0 \cdot x_0 \ +_{[n]} \ \cdots \ a_p \cdot x_p = b_0 \cdot y_0 \ +_{[n]} \ \cdots \ b_q \cdot y_q.$$

Using arithmetic modulo 2^n, we can easily isolate the most complex variable, say $z_{[m]}$, with coefficient c on the left-hand side. The resulting equation has the following form:

$$c \cdot z_{[m]} = d_0 \cdot w_{0[m_0]} \ +_{[n]} \ \cdots \ d_j \cdot w_{j[m_j]}.$$

Now, we must eliminate c in order to isolate z. If c is odd, we can do this by finding its multiplicative inverse, which is c^{2^k-1} for some $k \leq n-2$. We explain briefly why this is true. It is well-known [7] that the set of all positive integers less than and relatively prime to some positive integer p forms a group under multiplication modulo p, denoted $U(p)$. In particular, when $p = 2^n$, every odd positive integer less than p is in $U(p)$. Furthermore, $U(2^n)$ is isomorphic to the (external) direct product of the cyclic groups of order 2 and 2^{n-2}, which means that each element of $U(2^n)$ has order 2^k for some $k \leq 2^{n-2}$. Thus, if c is odd, there exists $k \leq n-2$ such that $c^{2^k} = 1$ modulo 2^n, and it follows that the inverse is c^{2^k-1}. To find the inverse i of c, we use the simple algorithm shown in Figure 1.

Thus, finding the inverse of c requires at most $2(n-2)$ n-bit multiplies, and assuming uniform distribution, the expected number of multiplies is about $2(n-3)$. In the examples we have done, however, the coefficient is almost always either 1 or $2^n - 1$, so the actual average number of multiplications is actually much less (between 0 and 2). After calculating the inverse, we simply multiply all terms

```
i := c;
while (c $\neq$ 1) do begin
  c := (c $\times$ c) mod 2^$n$;
  i := (i $\times$ c) mod 2^$n$;
end
```

Figure 1: Algorithm to find the multiplicitive inverse of c.

in the equation by the inverse and the resulting equation will have $z_{[m]}$ alone on the left-hand side.

Suppose on the other hand that c is even. Then we can write $c = 2^k \cdot b$ for some $k \geq 1$ and b odd. We can then split the equation into two equations as follows:

$$\begin{aligned} b \cdot z &= (d_0 \cdot w_0 \ +_{[n]} \ \cdots \ d_j \cdot w_j)[n-1:k] \quad \textbf{and} \\ 0_{[k]} &= d_0 \cdot w_0 \ +_{[k]} \ \cdots \ d_j \cdot w_j. \end{aligned}$$

The first equation can now be solved by calculating the inverse of b. Furthermore, though we will have to repeat the canonization process on the second equation, we have eliminated z from it without adding any variables, ensuring that the process will terminate.

Once we have an equation of the form

$$z_{[m]} = c_0 \cdot w_0 \ +_{[n]} \ \cdots \ c_j \cdot w_j,$$

there is one final step if $m < n$ (which is possible since, as mentioned in Table 1, we do not put any restrictions on the bit-widths of variables appearing in bitplus expressions). In this case, we know that the most significant bits of the right-hand side of the equation must be zero. So as above, we split it into two equations:

$$\begin{aligned} z_{[m]} &= c_0 \cdot w_0 \ +_{[m]} \ \cdots \ c_j \cdot w_j \quad \textbf{and} \\ 0_{[n-m]} &= (c_0 \cdot w_0 \ +_{[n]} \ \cdots \ c_j \cdot w_j)[n-1:m]. \end{aligned}$$

Again, we may have to canonize the second equation. Eventually, though, our initial equation will be transformed into a conjunction of equations, each solved for a different variable. We have found that it is very desirable for efficiency to transform these equations (via substitution) so that variables appearing on the left-hand side do not appear in the right-hand sides of any of the other equations.

4 Avoiding Bit-Slicing

As previously mentioned, whenever part of a bit-vector variable is extracted, that bit-vector is sliced into several parts, which reduces the level of abstraction and is thus to be avoided if possible. Our initial implementation of bit-vectors was based on the work done by Cyrluk et al. in [6]. However, our decision to use bitplus expressions as our internal representation was a significant departure from their method. As a result, we were able to increase the range of arithmetic examples which can be verified automatically.

However, it turns out that for some examples, even the original core theory of concatenation and extraction benefits from this change in the internal representation. This is because bit-slicing can be avoided in many cases. Suppose w is a bit-vector of size n and consider the following example:

$$w[n-1:1] = w[n-2:0] \quad \Rightarrow \quad w[n-1] = w[0]$$

In order to canonize this formula, the decision procedure of Cyrluk et al. generates a new variable for every bit of w,

$(x[0] = 0_{[1]})$	$\Rightarrow$	$((x[n-1:1]+1) \circ 0_{[1]})$	$=$	$x+2)$	(1)
$(x[0] = 1)$	$\Rightarrow$	$((x+(-1)_{[n]})[n-1:1]$	$=$	$x[n-1:1])$	(2)
		$\textbf{NOT}\ ((x+y+(-1)_{[n]})[0])$	$=$	$(x+y)[0]$	(3)
		$((2^n-2) \cdot x_{[n-1]} +_{[n]} (-1)_{[n]})[n-1:1]$	$=$	$\textbf{NOT}\ x_{[n-1]}$	(4)
		$(y_{[1]} \circ x_{[n-2]} +_{[n]} 1)[n-1]$	$=$	$(x_{[n-2]} +_{[n]} y_{[1]})[n-2]$	(5)

Figure 2: Bit-Vector Arithmetic Verification Examples. x and y are bit-vectors of size n unless otherwise specified.

resulting in a conjunction of n equations:

$$\begin{aligned} (w_{n-1} &= w_{n-2}) \ \wedge \\ (w_{n-2} &= w_{n-3}) \ \wedge \\ &\vdots \\ (w_1 &= w_0) \quad \wedge \\ (w &= w_{n-1} \circ w_{n-2} \cdots w_0) \end{aligned}$$

where each w_i is a new bit-vector of length one. Using our canonizer, only the most and least significant bits are sliced, resulting in the following two equations:

$$\begin{aligned} (2^{n-2} \cdot x_{[1]} +_{[n-1]} y_{[n-2]} &= 2 \cdot y_{[n-2]} +_{[n-1]} z_{[1]}) \ \wedge \\ (w &= x_{[1]} \circ y_{[n-2]} \circ z_{[1]}) \end{aligned}$$

We then invoke the solver and end up with:

$$\begin{aligned} (y_{[n-2]} &= (-1)_{[n-2]} \cdot z_{[1]}) \ \wedge \\ (x_{[1]} &= z_{[1]}) \qquad\quad \wedge \\ (w &= x_{[1]} \circ y_{[n-2]} \circ z_{[1]}) \end{aligned}$$

Thus, instead of producing n equations, we produce only three. The information from the other equations is stored in the coefficients. Most of w remains as an abstract bit-vector.

5 Results

The examples shown in Figure 2 demonstrate the kinds of formulas which can be verified using the methods described above. Formulas (1) through (3) are small pieces of much larger formulas from processor verification, and (4) and (5) are simply test benchmarks which we feel are representative of the complexity of the general problem. In general, SVC must solve many such small problems, as well as similar problems in other theories, as part of proving a larger formula.

Examples three through five are easily verified using only the SVC canonizer. These examples can also be verified using *BMDs (in contrast, the first two examples cannot be directly verified using only *BMDs because they include Boolean connectives). In Table 3, the column labeled "SVC" shows the time required to verify these examples running SVC on a 200 MHz Pentium Pro. The second column shows the time required to verify the same property on the same machine using the *BMD package from Bryant and Chen [3]. The third column shows the time required on a 300 MHz UltraSparc-30 using Laurent Arditi's *BMD implementation which has special support for bit-vector and Boolean operations [1]. For the two *BMD packages, the examples were run with four typical variable orderings and the best time for each example is reported. In order to better compare the two different *BMD implementations, a simple n-bit adder example was also done and the result is listed as example A.

Example	n	SVC	*BMD 1	*BMD 2
1	32	2	N/A	40
2	32	2	N/A	1100
3	8	2	440	30
3	16	2	265000	70
3	32	2	> 500000	180
4	8	2	112	80
4	16	2	26400	720
4	32	2	> 500000	8790
5	8	30	95	60
5	16	111	22700	390
5	32	520	> 500000	3780
A	32	0.2	32	80

Table 3: Results on Bit-Vector Arithmetic Verification Examples. Times are in milliseconds. Examples 1 through 5 are from Table 2. Example A is the expression $x_{[n]} + y_{[n]}$.

Several important observations can be made from these data. First, it is clear that *BMDs benefit greatly from the special-purpose algorithms in Arditi's implementation. From example A, which does not use the special-purpose algorithms, it also seems clear that Arditi's package could benefit further from implementation in a lower-level language (Bryant and Chen's package is in C, whereas Arditi's package is in Scheme). However, even given these possible improvements, SVC would still outperform *BMDs on all examples with the possible exception of example 5 (which we will come back to in a moment). More importantly, the performance of SVC is the same despite increasing bit width on two out of three of the examples. The reason for this is that SVC is able to maintain its bit-vector abstraction in these examples and thus does not need to consider each bit individually.

The reason SVC does poorly in example 5 is that it ends up slicing x and thus the execution time depends on the number of bits in x. However, if SVC were to split on y instead, it could avoid slicing x. This is a problem similar to variable ordering in *BMDs. We were able to write a slightly modified version of example 5 which forced SVC to split on y first. The result was an execution time of less than 10 ms independent of bit-width. One of the areas of ongoing research in SVC is how to automatically choose the best variables for case-splitting.

The primary application for SVC is microprocessor verification. As mentioned above, we are currently applying it to the TORCH microprocessor [11]. Table 4 shows the times required to verify several formulas from this effort. These formulas are large and require cooperating decision procedures from several theories including the bit-vector theory

Example	Size (KB)	Case Splits	Time (s)
PCUnitDataPath	29	1254	1.67
TakenBranch	31	38	0.15
IFetchControl	65	29676	24.1
IFetchPC	42	69374	317

Table 4: TORCH Microprocessor Verification Examples. Time is in seconds

(thus they cannot be verified using *BMDs alone and a direct comparison cannot be made). The first example, PCUnitDataPath, verifies that the program counter is calculated correctly. The second example, TakenBranch, verifies that the hardware correctly identifies when a conditional branch should be taken. The last two examples verify various properties of the instruction fetch unit. For each example, we show the size of the formula, the total number of case splits, and the time required to verify the formula. We would be unable to automatically verify these formulas without the bit-vector decision procedure.

6 Conclusions

Our method for dealing with bit-vector arithmetic has many advantages. First, it is complete and automatic, and although the complexity of the problem dictates that there are examples for which it will be exponentially slow, it is efficient on the examples we have encountered so far. Second, we reason at the word-level and avoid slicing bit-vectors whenever possible. This avoids some of the blow-ups in time and space experienced by other methods. Finally, our method is easy to implement and integrate into an environment of cooperating decision procedures such as that found in SVC.

As we have seen, one limitation of SVC is that the choice of which variable to split on can greatly influence its efficiency. Additionally, there is no way to represent non-linear multiplication of bit-vectors directly as there is with *BMDs. There is also currently no way to reason about bit-vectors of unknown size. We are working on addressing some of these limitations. For example, we have implemented a couple of automatic learning strategies for choosing splitters which dramatically increase the efficiency of many examples.

Some areas of future research include finding an efficient abstraction for Boolean bit-vector operations and finding more efficient ways of dealing with the overflow operator. We would also like to explore extensions to bit-vectors of unknown size and variable extraction indices, as well as non-linear arithmetic. We intend to continue to use SVC in our verification of TORCH and expect that it will continue to develop towards an important and powerful tool for automatic hardware verification.

Acknowledgments

We would like to thank Yirng-An Chen for his help in understanding and using *BMDs. We would like to thank other members of the TORCH verification effort for their work in specifying and translating examples for this paper: Jeffrey Su, Jens Skakkebæk, and especially Laurent Arditi who also provided insight and analysis for comparison with *BMDs. Also, we would like to thank the group at SRI International for valuable discussions, especially David Cyrluk whose work on bit-vectors provided a valuable starting point. This work was sponsored by DARPA contract number DABT63-96-C-0097 and by a National Defense Science and Engineering Graduate Fellowship. The content of this paper does not necessarily reflect the position of the policy of the government and no official endorsement should be inferred.

References

[1] Laurent Arditi. *BMDs Can Delay the Use of Theorem Proving for Verifying Arithmetic Assembly Instructions. In Srivas [12], pages 34–48.

[2] C. W. Barrett, D. L. Dill, and J. R. Levitt. Validity Checking for Combinations of Theories with Equality. In Srivas [12], pages 187–201.

[3] Randal E. Bryant and Yirng-An Chen. Verification of Arithmetic Circuits with Binary Moment Diagrams. In 32^{nd} *ACM/IEEE Design Automation Conference*, pages 535–541, San Francisco, CA (USA), 1995.

[4] J. R. Burch and D. L. Dill. Automatic Verification of Microprocessor Control. In Dill, editor, *Computer-Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 68–79, Stanford, CA (USA), June 1994.

[5] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's Decision Procedure for Combinations of Theories. In 13^{th} *International Conference on Automated Deduction*, pages 463–477, Rutgers University, NJ (USA), July 1996.

[6] D. Cyrluk, O. Möller, and H. Rueß. An Efficient Decision Procedure for the Theory of Fixed-Sized Bitvectors. In 9^{th} *International Conference on Computer-Aided Verification*, 1997.

[7] Joseph A. Gallian. *Contemporary Abstract Algebra*. D. C. Heath and Company, second edition, 1990.

[8] Warren A. Hunt Jr. Microprocessor Design Verification. *Journal of Automated Reasoning*, 5(4), December 1989.

[9] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient Validity Checking for Processor Verification. In *IEEE Internationl Conference on Computer-Aided Design*, pages 2–6, San Jose, CA (USA), November 1995. IEEE Computer Society Press.

[10] Leo G. Marcus. *SDVS 13 Users' Manual*. The Aerospace Corporation, El Segundo, CA 90245, October 1994. Aerospace Report ATR-94(4778)-5.

[11] M. Smith, M. Lam, and M. Horowitz. Boosting Beyond Static Scheduling in a Superscalar Processor. In *International Symposium on Computer Architecture*, pages 344–354, Seattle, WA, May 1990. IEEE/ACM.

[12] Srivas, editor. *International Conference on Formal Methods in Computer-Aided Design*, volume 818 of *Lecture Notes in Computer Science*, Palo Alto, CA (USA), November 1996. Springer-Verlag.

[13] Wai Wong. Modelling Bit Vectors in HOL: the word Library. In Joyce and Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 371–384, Vancouver, Canada, August 1993.